Discovering Model Transformation Pre-conditions using Automatically Generated Test Models

Jean-Marie Mottu
Université de Nantes, LINA
Nantes, France
Email: mottu-jm@univ-nantes.fr

Sagar Sen
Simula Research Laboratory
PB 134, 1325 Lysaker, Norway
Email: sagar@simula.no

Juan Cadavid
CEA, LIST
Gif-sur-Yvette, France
Email: Juan.Cadavid@cea.fr

Benoit Baudry
INRIA
Rennes, France
Email: Benoit.Baudry@inria.fr

***Abstract*—Specifying a model transformation is challenging as it must be able to give a meaningful output for any input model in a possibly infinite modeling domain. Transformation pre-conditions constrain the input domain by rejecting input models that are not meant to be transformed by a model transformation. This paper presents a systematic approach to discover such pre-conditions when it is hard for a human developer to foresee complex graphs of objects that are not meant to be transformed. The approach is based on systematically generating a finite number of test models using our tool, PRAMANAto first cover the input domain based on input domain partitioning. Tracing a transformation's execution reveals why some pre-conditions are missing. Using a benchmark transformation from simplified UML class diagram models to RDBMS models we discover new pre-conditions that were not initially specified.**

Keywords: Model transformation, Testing, Model Generation, Pre-condition, Incomplete Domain Specification

I. INTRODUCTION

Model transformations are core *Model Driven Engineering* (MDE) components that automate important steps in software development such as refinement of an input model, refactoring to improve maintainability or readability of the input model, aspect weaving into models, exogenous/endogenous transformations of models, and the classical generation of text such as code from models. Transformations are different from general-purpose languages such as Java as they raise the level of abstraction to facilitate the processing of models which are *graphs of interconnected objects* specified by an input metamodel. Examples of transformation languages include those based on graph rewriting [5], imperative execution (e.g. Kermeta [27]), and rule-based transformation (e.g. ATL [18]).

Testing model transformations presents several new challenges [6]. Model transformation testing ensures that the implementation is correct w.r.t a specification. The tester should first consider the correctness of the specification, i.e. the transformation produces the expected output model from a given input model. In this paper, we consider the case in which the specification is possibly incomplete, i.e. there are *missing pre-conditions*. The execution of such an incompletely specified transformation can lead to its computation not finishing or producing incorrect output models. Ideally we must specify a model transformation such that it can appropriately handle *all models* in its input modeling domain. This means that a model transformation must correctly transform models that it is supposed to transform and reject those it is not designed to transform. For instance, the input domain for an *object persistence* transformation [7] comprises of UML class models (UMLCD) and output domain of entity-relationship (RDBMS) models. Classes must have at least one primary attribute (an unique identifier) for it to be transformed to entities in the RDBMS model. Any model in the input domain with a class that does not have a primary attribute is not eligible for transformation and must be rejected.

A model transformation's specification entails *pre-conditions* as contracts on the input domain of a model transformation to prevent it from processing ineligible models. However, pre-conditions can be *notoriously hard* to foresee and specify for a model transformation given that the input domain is a set of infinite objects (for example, due to 0..* multiplicities in a metamodel) with **potentially infinite ways of being interconnected**. Our hypothesis is that a finite representative subset of models in the input domain can help improve the specification of a model transformation's pre-condition. How can we automatically generate this finite set of models and consequently discover new pre-conditions for a model transformation? This is the question that intrigues us and we present an approach to address it.

Incremental Pre-condition Discovery: Our approach consists of (1) Automatic generation of a finite set of input test models *I* for a transformation based on *input domain partitioning* in the *current specification* (consisting of input metamodel, invariants on it, and initial set of pre-conditions) of the input domain. We use the tool PRAMANA based on previous work [30] that solves an ALLOY specification *A* of the input domain to generate these models. (2) We dynamically analyze the execution of a transformation using test models in *I*. Test models that either run into an *infinite loop* transformation resulting in a *stack overflow*, or, transform to an output model *not in the specified output domain* are deemed as *failed inputs*. Either those failed inputs are out of the input domain, or the transformation is unable to correctly transform them. In both cases, the specification is possibly incomplete, and it should be completed with new pre-conditions. For every failed input we produce an execution trace to identify the modeling pattern (objects and relationships) that leads to failure. (3) We introduce a human-in-the-loop to transform the modeling patterns to new pre-conditions and repeat the generation and test process until all generated test models in are executed correctly by the transformation. New pre-conditions are represented as new ALLOY facts in *A*. They are also represented by the human expert as OCL constraints (the classical constraint modeling language) to improve the specification of the input domain.

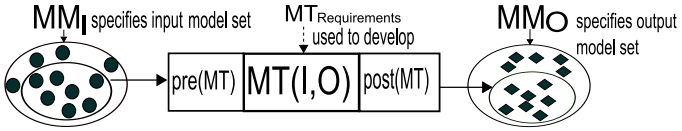


Fig. 1. A Model Transformation

We apply our approach to discover new pre-conditions for the benchmark model transformation of simplified Unified Modelling Language Class Diagram (UMLCD) to Relational Database Management Systems (RDBMS) models called class2rdbms. The transformation class2rdbms was initially specified by a panel of experts at the MTIP workshop in Models 2005 [7] with one pre-condition. We discover *12 new pre-conditions* for class2rdbms. This discovery of knowledge can be seen as valuable gems added to the initial specification of class2rdbms. The new pre-conditions also illustrate the fact that some modeling patterns have a potentially complex structures that cannot be foreseen by a human modeler. For instance, we discovered a pre-condition that rejects an input model for class2rdbms because it contains a class A which inherits from a persistent class B and has an attribute with the same name and type of an attribute in persistent class B. A model like this cannot be persisted into an RDBMS model because the columns will conflict. The discovery of pre-conditions was based on executing upto 13 series of 1890 automatically generated test models in each iteration and several hours of computation on two high-end computers.

We summarize our contributions as follows:
Contribution 1: A systematic approach to discover new model transformation pre-conditions
Contribution 2: Discovery of a set of 12 new pre-conditions for the benchmark transformation class2rdbms [7].

The paper is organized as follows. In Section II we present the transformation case study class2rdbms as a running example and we use it to describe the problem. In Section III, we present foundational ideas to understand the paper. In Section IV, we present our approach for pre-condition discovery based on automatic test model generation and input domain partitioning. In Section V, we present the experimental setup to generate models and present the discovered pre-conditions. In Section VII we present threats to validity. In Section VII we present related work. We conclude in Section VIII.

II. Case Study and Motivation

Our general objective is to discover novel pre-conditions $pre(MT)$ for a model transformation $MT(I,O)$. $MT(I,O)$ is a program applied on a set of input models I to produce a set of output models O as illustrated in Figure 1. The set of all input models is specified by a metamodel MM_I (For example, simplified UMLCD in Figure 2). The set of all output models is specified by metamodel MM_O. Post-conditions $post(MT)$ constrains $MT(I,O)$ to producing a subset of all possible output models. The model transformation is developed based on a set of textual specification of requirements $MT_{Requirements}$.

A. Transformation Case Study

Our case study is the transformation from simplified UML Class Diagram models to RDBMS models called class2rdbms. In this section we briefly describe class2rdbms.

In Figure 2, we present the simplified UMLCD input metamodel for class2rdbms. The concepts and relationships in the input metamodel are stored as an Ecore model [10] (Figure 2 (a)). **Four invariants** among a total of **ten** on the simplified UMLCD Ecore model, expressed in Object Constraint Language (OCL) [29], are shown in Figure 2 (b). The Ecore model and the invariants together represent the input domain for class2rdbms. The Ecore and OCL are industry standards used to develop metamodels and specify different invariants on them. OCL is not a domain-specific language to specify invariants. However, it is designed to formally encode natural language requirements specifications independent of its domain.

The input metamodel MM_I gives an initial specification of the input domain. In addition, the model transformation itself has a set of pre-conditions $pre(MT)$ that input models need to satisfy to be correctly processed. Constraints in the pre-condition for class2rdbms include: (a) All Class objects must have at least one primary Property object. (b) The type of a Property object can be a Class C, but finally the transitive closure of the type of Property objects of Class C must end with type PrimitiveDataType. In our case we approximate this recursive closure constraint by stating that Property object can be of type Class up to a depth of 3 and the 4th time it should have a type PrimitiveDataType. This is a finitization operation to avoid navigation in an infinite loop. (c) There are no associations between class in an inheritance hierarchy. The first pre-condition (a) was in the initial specification of MTIP [7] and the two other ones (b and c) have been added when the transformation has been implemented and previously studied. We use the implementation written in Kermeta. Kermeta [20] is a language for specifying metamodels, models, and model transformations that are compliant to the Meta Object Facility (MOF) standard [28].

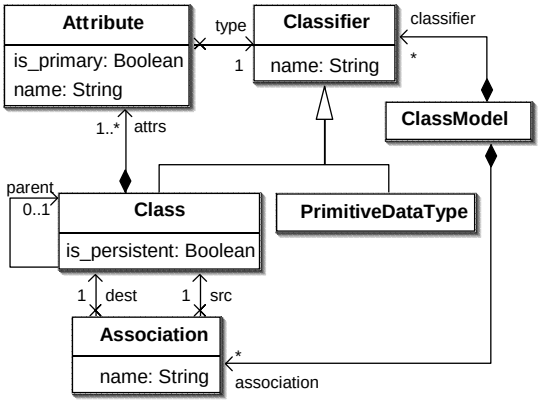
We choose class2rdbms as our representative case study to address our problem of pre-condition discovery. It serves as a sufficient case study for several reasons. The transformation is the benchmark proposed in the MTIP workshop at the MoDELS 2005 conference [7] to experiment and validate model transformation language features. The input domain metamodel of simplified UMLCD covers all major metamodeling concepts such as inheritance, composition, finite and infinite multiplicities. The constraints on the simplified UMLCD metamodel contain both first-order and higher-order constraints. There also exists a constraint to test transitive closure properties on the input model such as there must be no cyclic inheritance (Figure 2 (b)). class2rdbms is not just a refactoring, it is an exogenous transformation: input and output metamodels are different. The class2rdbms exercises most major model transformation operators such as navigation, creation, and filtering (described in more detail in [25]) enabling us to find loops due to several transformation operators. Among the limitations the simplified UMLCD metamodel does not contain Integer and Float attributes. There are also no inter-metamodel references in the metamodel.

B. Motivation

Any computer program has limitations on what it can process: e.g. a division of integers is not supposed to divide per

zero. Programs have a limitation on memory and computation capacity for any given algorithm. Design by contracts [16], [23] is a programming paradigm that emerged at the turn of the 21st century as an effective and practical means to prevent programs from processing faulty inputs and producing faulty outputs due to specification of pre and post conditions. However, specifying contracts from model transformations in the form of pre-conditions is more complex. The input domain of a model transformation for instance is a set of potentially infinite graphs of interconnected objects (for instance, due to multiplicities 0..*). It is hard to foresee graph like patterns that could crash a model transformation and subsequently become pre-conditions.

The specification provided in [7] specifies how several structures in the models have to be transformed. For instance, it is specified that the "recursive nature of the drilling down" should be considered. Therefore, the initial specification contains a pre-condition (b) ensuring that the type of a Property object can be a Class C, but finally the transitive closure of the type of Property objects of Class C must end with type PrimitiveDataType. And the model transformation is able to process models with such structures. However, some model structures allowed by the input metamodel are not specified in the class2rdbms specification. The consequence is that the implementation of class2rdbms fails processing them. For instance, it fails when attributes are typed with their containment class. This failure leads the implementation of class2rdbms to loop infinitely. Therefore, we have to complete the specification to consider that structure (illustrating Figure 3), for instance preventing it in the input models.



(a) **Ecore Meta-model**

context Class

inv noCyclicInheritance: not self.allParents()->includes(self)
inv uniqueAttributesName: self.attrs->forAll(att1, att2 |
att1.name=att2.name implies att1=att2)

context ClassModel

inv uniqueClassifierNames: self.classifier->forAll(c1, c2 |
c1.name=c2.name implies c1=c2)

inv uniqueClassAssociationSourceName :
self.association->forAll(ass1, ass2 | ass1.name=ass2.name
implies (ass1=ass2 or ass1.src != ass2.src))

(b) **OCL Invariants**

Fig. 2. (a) Simplified Class Diagram Ecore Meta-model (b) OCL constraints on the Ecore metamodel

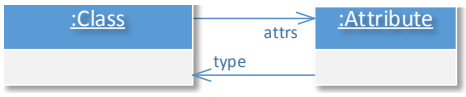


Fig. 3. Input model structure unspecified in the class2rdbms specification

Another motivating sample with class2rdbms is that an input model could be transformed into an output model with two identical columns which is incorrect as it violates an output model invariant.

Our challenge is to *focus on discovering pre-conditions* that ensure that:

- A model transformation does not *loop infinitely* and consequently leading to a *stack overflow*.
- Output models are correct, i.e., they always conform to the output metamodel and do not violate a post-condition in the set $post(MT)$.

III. Foundations

This section presents foundational ideas used by the methodology for generating models to discover pre-conditions in Section IV. We briefly describe PRAMANA for automatic test model generation in Section III-A. Test model generation in this paper is guided by coverage criteria based testing strategies, which are input domain independent. These testing strategies are described in Section III-B.

A. PRAMANA: *A Tool for Automatic Model Generation*

We use the tool PRAMANA previously introduced (with the name CARTIER) in our paper [30] to automatically generate test input models (*test models* in the rest of the paper since we do not consider output models). PRAMANA transforms the input domain specification of a model transformation to a common constraint language ALLOY. Solving the ALLOY model gives zero or more models in the input domain of a transformation. PRAMANA first transforms a model transformation's input metamodel expressed in the Eclipse Modeling Framework [10] format called Ecore using the transformation rules presented in [30] to ALLOY. Basically, classes in the input metamodel are transformed to ALLOY signatures and implicit constraints such as inheritance, opposite properties, and multiplicity constraints are transformed to ALLOY facts.

Second, PRAMANA also addresses the issue of transforming invariants on metamodels and pre-conditions expressed in Object Constraint Language (OCL) to ALLOY. The automatic transformation of OCL to ALLOY presents a number of challenges that are discussed in [1]. We do not claim that all OCL constraints can be manually/automatically transformed to ALLOY for our approach to be applicable in the most general case. The reason being that OCL and ALLOY were designed with different goals. OCL is used mainly to query a model and check if certain invariants are satisfied. ALLOY facts and predicates on the other hand enforce constraints on a model. This is in contrast with the side-effect free OCL. The core of ALLOY is declarative and is based on first-order relational

TABLE I. MAPPINGS FROM OCL TO ALLOY

Let v be a variable, col a collection, $expr$ an expression, be an expression that returns a boolean value, o an expression that returns an object, T a type, $propertyCallExpr$ an expression invoking a property on an object

OCL Expression Type	ALLOY Abstract Syntax Type
$context\ T\ inv\ expr$	$sig\ T\{...\}\{expr\}$
$col \rightarrow forAll(v:T \mid be)$	$all\ v:T \mid be$
$col \rightarrow forAll(v:col \mid be)$	$all\ v:col \mid be$
$expr1 and expr2$	$expr1\ \&\&\ expr2$
$expr1 or expr2$	$expr1 \mid\mid expr2$
$not\ be$	$!be$
$col \rightarrow size()$	$\#col$
$col \rightarrow includes(o:T)$	$o\ in\ col$
$col \rightarrow excludes(o:T)$	$o\ !in\ col$
$col1 \rightarrow includesAll(col2)$	$col2\ in\ col1$
$col1 \rightarrow excludesAll(col2)$	$col2\ !in\ col1$
$col \rightarrow including(o:T)$	$col\ +\ o$
$col \rightarrow excluding(o:T)$	$col\ -\ o$
$col \rightarrow isEmpty()$	$no\ col$
$col \rightarrow notEmpty()$	$some\ col$
$expr.propertyCallExpr$	$expr.propertyCallExpr$
$if\ be\ then\ expr1\ else\ expr2$	$be \Rightarrow expr1\ else\ expr2$
$expr.oclIsUndefined$	$\#expr = 0$
$expr \rightarrow oclIsKindOf(o:T)$	$expr\ in\ o$
$col1 \rightarrow union(col2)$	$col1 + col2$
$col1 \rightarrow intersection(col2)$	$col1\ \&\ col2$
$col1 \rightarrow product(col2)$	$col1 \rightarrow col2$
$col \rightarrow sum()$	$sum\ col$
$col1 \rightarrow symmetricDifference(col2)$	$(col1 + col2) - (col1 \& col2)$
$col \rightarrow select(be)$	$v:col \mid be$
$col \rightarrow isUnique(propertyCallExpr)$	$no\ disj\ v1, v2 : col \mid v1.propertyCallExpr = v2.propertyCallExpr$

logic with quantifiers while OCL includes higher-order logic and has imperative constructs to call operations and messages making some parts of OCL more expressive. In our case study, we have been successful in transforming all meta-constraints on the UMLCD metamodel to ALLOY from their original OCL specifications. Nevertheless, we are aware of OCL's status as a current industrial standard and thus provide an automatic mapping to complement our approach.

Previous work exists in mapping OCL to ALLOY. The tool UML2Alloy [1] takes as input UML class models with OCL constraints. The authors present a set of mappings between OCL collection operations and their ALLOY equivalents. Here we present our version of such transformation derived from [1] and written in Kermeta.

The context of an OCL constraint (which is what defines the value of the *self* function) determines the place of the constraint within the generated ALLOY model. It is added as an appended fact. The mappings in Table I (taken in part from [1]) show the set of transformation rules that can be implemented. OCL constraints in this article were transformed manually to ALLOY due to their complexity.

However, some classes of OCL invariants cannot be automatically transformed to ALLOY using the simple rules in Table I. For example, consider the invariant for no cyclic inheritance in Figure 2(b) [4]. The constraint is specified as the fact in Listing 1. This is an example in which the richness of the ALLOY language overcomes OCL - it is not possible to specify this constraint in OCL without using recursive queries since there is no transitive closure operator.

```
fact noCyclicInheritance {
  no c: Class | c in c.^parent
}
```

Listing 1. ALLOY Fact for No Cyclic Inheritance

B. Test Selection Strategies

We guide automatic model generation to select test models that cover the input domain of a model transformation following our previous works [15], [30]. We define a strategy as a process that generates ALLOY *predicates* which are constraints added to the ALLOY model synthesized by PRAMANA as described in Section IV. This combined ALLOY model is solved and the solutions are transformed to model instances of the input metamodel that satisfy the predicate. We guide model generation based on **input-domain partition based strategies** where we combine *partitions* on domains of all properties of a metamodel (cardinality of references and domain of primitive types for attributes). A *partition* of a set of elements is a collection of n ranges A_1,..., A_n such that A_1, ..., A_n do not overlap and the union of all subsets forms the initial set. These subsets are called *ranges*. We use partitions of the input domain since the number of models in the domain are infinitely many. Using partitions of the properties of a metamodel we define two coverage criteria that are based on different strategies for combining partitions of properties. Each criterion defines a set of *model fragments* for an input metamodel. These fragments are transformed to predicates on metamodel properties by PRAMANA. For a set of test models to cover the input domain at least one model in the set must cover each of these model fragments. We generate model fragment predicates using the following coverage criteria to combine partitions:

- **AllRanges Criteria:** AllRanges specifies that each range in the partition of each property must be covered by at least one test model.
- **AllPartitions Criteria:** AllPartitions specifies that the whole partition of each property must be covered by at least one test model.

The notion of coverage criteria to generate model fragments was initially proposed in our paper [15]. The accompanying tool called Meta-model Coverage Checker (MMCC) [15] generates model fragments using different test criteria taking any metamodel as input. Then, the tool automatically computes the coverage of a set of test models according to the generated model fragments. If some fragments are not covered, then the set of test models should be improved in order to reach a better coverage.

In this paper, we use the model fragments generated by MMCC for the UMLCD Ecore model (Figure 2). We use the criteria AllRanges and AllPartitions. For example, in Table II, *mfAllRanges1* and *mfAllRanges2* are model fragments generated by PRAMANA using MMCC for the *name* property of a classifier object. The *mfAllRanges1* states that there must be at least one classifier object with an empty name while *mfAllRanges2* states that there must be at least one classifier object with a non-empty name. These values for name are the ranges for the property. The model fragments chosen using AllRanges *mfAllRanges1* and *mfAllRanges2* define two partitions *partition1* and *partition2*. The model fragment *mfAllPartitions1* chosen using AllPartitions defines both *partition1* and *partition2*. The Table II lists the 27 consistent model fragments used in our experiments. Several model fragments are inconsistent with respect to the input metamodel and other fragments. In [26] we discuss why fragments such as *mfAllRanges7* are inconsistent.

TABLE II. CONSISTENT MODEL FRAGMENTS GENERATED USING ALLRANGES AND ALLPARTITIONS STRATEGIES

Model-Fragment	Description
mfAllRanges1	A Classifier $c \mid c.name =$""
mfAllRanges2	A Classifier $c \mid c.name! =$""
mfAllRanges3	A Class $c \mid c.is\_persistent = True$
mfAllRanges4	A Class $c \mid c.is\_persistent = False$
mfAllRanges5	A Class $c \mid \#c.parent = 0$
mfAllRanges6	A Class $c \mid \#c.parent = 1$
mfAllRanges8	A Class $c \mid \#c.attrs = 1$
mfAllRanges9	A Class $c \mid \#c.attrs > 1$
mfAllRanges10	An Attribute $a \mid a.is\_primary = True$
mfAllRanges11	An Attribute $a \mid a.is\_primary = False$
mfAllRanges12	An Attribute $a \mid a.name =$""
mfAllRanges13	An Attribute $a \mid a.name! =$""
mfAllRanges14	An Attribute $a \mid \#a.type = 1$
mfAllRanges15	An Association $as \mid as.name =$""
mfAllRanges16	An Association $as \mid as.name! =$""
mfAllRanges17	An Association $as \mid \#as.dest = 1$
mfAllRanges18	An Association $as \mid \#as.src = 1$
mfAllRanges24	An ClassModel $cm \mid \#cm.association > 1$
mfAllPartitions1	Classifiers $c1,c2 \mid c1.name =$"" and $c2.name! =$""
mfAllPartitions2	Classes $c1,c2 \mid c1.is\_persistent = True$ and $c2.is\_persistent = False$
mfAllPartitions3	Classes $c1,c2 \mid \#c1.parent = 0$ and $\#c2.parent = 1$
mfAllPartitions5	Attributes $a1,a2 \mid a1.is\_primary = True$ and $a2.is\_primary = False$
mfAllPartitions6	Attributes $a1,a2 \mid a1.name =$"" and $a2.name! =$""
mfAllPartitions7	An Attribute $a \mid \#a.type = 1$
mfAllPartitions8	Associations $as1,as2 \mid as1.name =$"" and $as2.name! =$""
mfAllPartitions9	An Association $as \mid \#as.dest = 1$
mfAllPartitions10	An Association $as \mid \#as.src = 1$

These model fragments are transformed to ALLOY predicates by PRAMANA. For instance, model fragment *mfAllRanges8* is transformed to the predicate in Listing 2.

```
pred mfAllRanges8 {
  some c : Class | #c.attrs=1
}
```

Listing 2. ALLOY Predicate for *mfAllRanges8*

IV. APPROACH

In this section we present a methodology to discover pre-conditions for a given model transformation. We use class2rdbms as a running example to illustrate the methodology. The process is divided into 8 steps as illustrated in Figure 5. Section IV-A describes how we detect that an input model is outside the input domain of the model transformation (in steps 3 and 4). The Section IV-B describes the identification of *incorrect input excerpts* based on *transformation traces* (in steps 2, 5, and 6). The Section IV-C describes how the tester can write a new pre-condition (in step 8) based on *incorrect input model pattern* obtained by generalizing a set of incorrect input excerpts (in step 7).

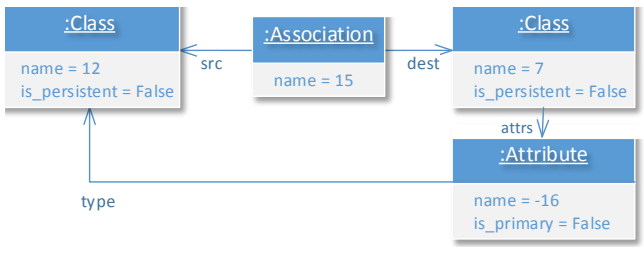


Fig. 4. Sample of an input model excerpt causing transformation loop

A. Detecting incorrect input models

We identify two reasons why an input model should not be selected as a model transformation entry: when the transformation is unable to transform it, and when it is transformed into an incorrect output model.

1) Non-transformable input model (Step 3): In Step 3, of Figure 5, we find out why an input model is non-transformable. For this, we consider a fundamental characteristic of model transformations: they are designed to navigate through the model and find modeling elements to transform. *Navigation* is one of the three main operations performed by a model transformation [25]. When it navigates a model it can enter into navigation loops and no output will never be returned. The transformation can run infinitely but most of the time it will crash (in particular once the memory stack is full, or if a time limit is reached). We detect that a transformation loops when our experiments return stack overflow errors.

In the Figure 4, we present a model excerpt extracted from an input model that loops when executed by class2rdbms. The transformation hence never returns an output model. This input model contains an *association* named "15" (names in generated models have a string type but have an integer value to simplify solving with Alloy) which is supposed to be transformed into a *column* in an output RDBMS model. This *association* has a destination *non persistent class* named "7". The attributes of this class need to be transformed into columns in the output RDBMS model. The *attribute* of this class "7" named "-16", is supposed to be transformed to a *column* typed with the *non persistent class* named "12" which in turn is the source class for the *association* named "15". The Figure 4 illustrates a closed loop and the transformation will infinitely navigate via the *association* named "12". Loops such as this will not allow the transformation class2rdbms to terminate.

Identifying the problematic input excerpt is complicated as a tester should identify the model excerpt that loops in a potentially large input model. This particular excerpt in Figure 4 contains 4 objects (2 classes, 1 association, and 1 attribute). However, the excerpt was found in an input model containing 50 objects and their properties (names, etc.).

2) Transformed into an incorrect output model (Step 4): Output models are incorrect when they do not conform to the output metamodel+invariants or when they do not satisfy the transformation's post-conditions. For instance, an RDBMS model should not have a table with several identical columns (same name and type). This is an invariant that must be satisfied by any RDBMS model. This invariant, written in OCL, is:

```
context Table
cols->forAll(c1, c2 | (c1.name = c2.name
    and c1.type = c2.type) implies c1=c2)
```

When an input model produces an incorrect output model, the constraint checking mechanism in Kermeta identifies the incorrect part in the output model by raising an exception. For instance, if an input model CD is transformed to an RDBMS model with two identical columns with the same name and type then it will violate the post-condition above.

In the Figure 6, we present the incorrect excerpt from an output model. Both columns have identical names and

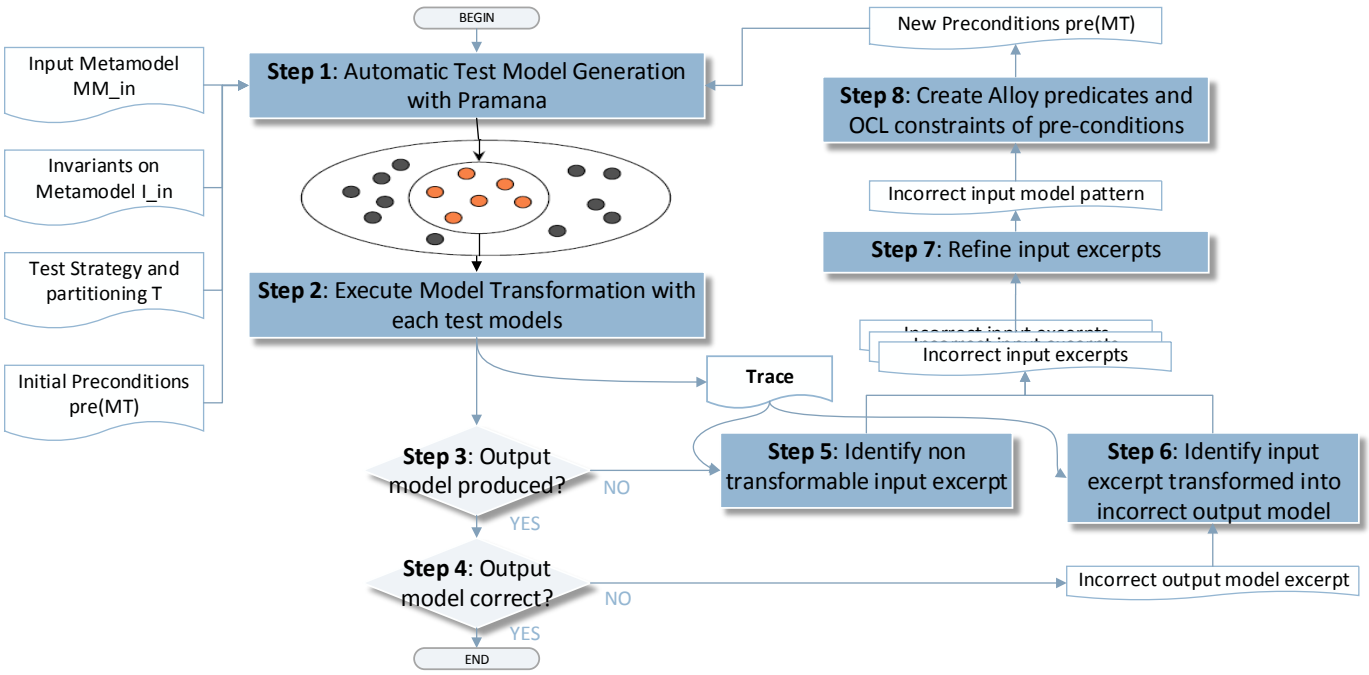


Fig. 5. Methodology for Pre-condition Discovery

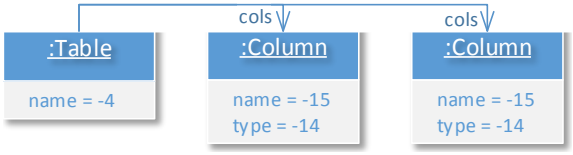


Fig. 6. Incorrect excerpt of an output model

types. This incorrect excerpt has been identified when the invariant is unsatisfied on an RDBMS model returned by the transformation.

The difficulty to create a pre-condition from an incorrect output model is harder than the previous point in Section IV-A1 since we have to consider an output excerpt. This excerpt has only 3 objects, whereas the entire output model has 67 objects and its input model has 40 objects with their properties making it hard to discover the problematic excerpt.

When an input model (i) cannot be transformed or when (ii) its output model is incorrect, we know that the input model is incorrect and the model transformation's specification should prevent it from being processed. At the end of those steps 3 and 4, we identify incorrect input models, and we know when the output model is produced which output excerpt is incorrect. But in both cases, we still have to study entire incorrect input models (at this point due to space limitations we use the Figure 4 to illustrate only an excerpt of an entire incorrect input model; entire input models are referred in the appendix A)

B. Identifying incorrect input excerpt

Incorrect input models are identified in the previous Section IV-A. We need to precisely identify the concerned elements in each incorrect input model. Finding these *incorrect input excerpts* can be hard as these models can be quite large. We will use them as an *incorrect input excerpt* to create new pre-conditions. That identification is made easier thanks to *traceability* in a model transformation.

1) Creating Trace (Step 2): While executing the model transformation, we collect a trace linking input elements with output elements. We also collect the transformation rules involved in each link, but it is not used in this work for the moment.

Various traceability approaches have been developed, but they are dedicated to a specific transformation language e.g. [14] or they take into account only classes and not their attributes [17]. A traceability approach has been developed [2] that is transformation language independent. We successfully used it previously considering test model improvement [3]. Each creation/modification of an element by a *rule* leads to the creation of a unique traceability link. Each link is a relation of three sets of elements referring to (i) a set of source elements (attribute or class instances), (ii) a set of target elements and (iii) the transformation rule that leads to this creation/modification. Each link corresponds to an execution of a rule, that may be executed several times on different input elements. Figure 8 illustrates an example. *Link*1 indicates that the output instance of *Table* has been created from one input instance of *Class*. Moreover, *Link*1 specifies that the instances it binds have been read and created by the *transform* rule.

Each trace captures relations between input/output models and the transformation. Traces can be analyzed independently and are not transformation language dependent.

2) Identifying non-executable input excerpt (Step 5): The trace produced from a non-executable input model is incomplete since no output model is produced. At the moment, we model a trace from a textual output of the trace in the

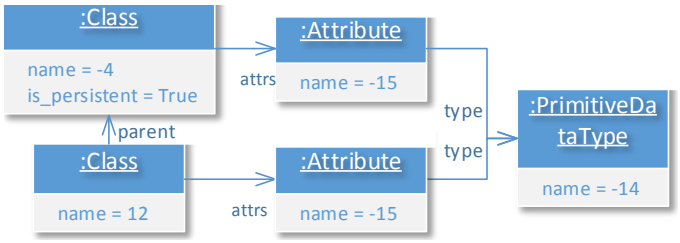


Fig. 7. Excerpt of an input model transformed into an incorrect output model (whose an excerpt is illustrated Figure 6)

console of the model transformation (such as the one of the Figure 8) to obtain an analyzable trace model. In this trace, we identify the loop, which is a series of identical links. These links refer the input model elements which are involved in the loop. Therefore, we can extract from the non transformable input model an excerpt which is incorrect, such as the one illustrated Figure 4.

3) Identifying out of the scope input excerpt (Step 6): Combining the trace and the incorrect excerpt of an output model, we identify incorrect excerpt in the input model. Input excerpt of the Figure 7 corresponds to the output excerpt of the Figure 6.

C. Creating new pre-condition

Once we detect one or more *input excerpt*, we generalize from it an *input pattern* which is finally used to create a pre-condition.

1) Generalizing input excerpts identifying incorrect input pattern (Step 7): The input excerpts are extracted from concrete input models that may contain several other elements not all useful to help us make new pre-conditions. For instance, the values of *name* in the Figure 4 are not important: the transformation will still loop even if they are different. In other excerpts, the values of *name* are important like in the sample of the Figure 6: columns are identical when their names are identical (and their types are identical).

There are many properties in a model, therefore the excerpts could be all different with potentially *one unique incorrect excerpt per incorrect model*. We will see in the next section V that they can be numerous input excerpts. Moreover, it would be ineffective to create a pre-condition for each one of them. For example, considering the excerpt of the Figure 7, it is useless to create a pre-condition preventing the creating of a model with a *class* exactly named "-4", with an *attribute* exactly named "-15", and so on.

Therefore, we generalize the *input excerpts*, rejecting useless properties and producing *input patterns*. For this, we develop the two-pass step 7. The **first pass** creates a pattern following two methods, the second pass controls pattern *accuracy* since too many properties could be rejected.

First method: We merge similar input excerpts keeping their similarities and rejecting their differences. We select subset of input excerpts to be merged based on the class and their assembly. We can use a tool such as EMF Compare which can be specialized to compare only what we are interested in. Then each set of similar excerpts is merged, the result is a *generalized pattern*.

Second method: When an input excerpt doesn't have similar excerpts to be merged with, we generate additional models. Since the objective here is to reject useless properties, we should compare several models with different properties' values. We apply the same technique than when generating test models at the step 1. Partitioning each properties domain, we generate additional input models. Therefore they are transformed, and those which are incorrect are analyzed identifying incorrect excerpts to be merged following the first method of the previous paragraph.

For instance, we produce the *incorrect input pattern* illustrated Figure 9 merging the excerpts of the Figure 4 with similar incorrect patterns. We keep only the two useful properties $is\_persistent = False$: they are always false in the merged excerpts. We identify that *names* and *is_primary* properties should not been considered since they differ in the different model excerpts which have been refined creating this input pattern.

Once we get an incorrect pattern, we should control its accuracy in a **second pass**. It is accurate when it contains all the necessary properties to create an effective pre-condition. Indeed, it is possible that merging rejects too many properties. One reason is that the number of generated models is limited, then useful properties combinations can be missing. Moreover, correctness of a model can depend on several properties of one model at the same time This is the case in the excerpt of the Figure 7: RDBMS invariant is violated only if both names of two different attributes are equal. However, merging this excerpt with similar excerpts would reject the properties *name* of *Attribute*, since their values are not equal between different models randomly generated per PRAMANA.

Therefore, we introduce a second pass which systematically modified in the incorrect input model the values of the properties that were in the excerpt and are no more in the pattern. The altered input models are executed, and step 3 and 4 are used to control if they are still incorrect models. Otherwise it means that the pattern is not accurate, and needs improvements.

This control is useful with the excerpt of the Figure 7. In that case, at the first pass, the merged pattern won't consider that the attributes' names in a model should be equals. During the control pass, the top attribute name is changed into "-15aa", and the bottom attribute name is changed into "-15ab". Therefore, the output model satisfies the invariant, and the input model is not incorrect anymore. A human can easily understand the problem reading the output invariant, and correct the pattern as illustrated Figure 10.

2) Writing new pre-condition (step 8): The incorrect input pattern is *manually re-written* to an ALLOY predicate. For instance, writing the predicate for the pre-condition G1, from the input pattern of the Figure 9:

```
fact noChainAssoFromClassTypeLevel1{
all c1:Class, assoc1:Association |
all a1:c1.attrs | (a1.type == assoc1.src and
    assoc1.dest == c1) => (c1.is_persistent=
    True or a1.type.is_persistent=True)
}
```

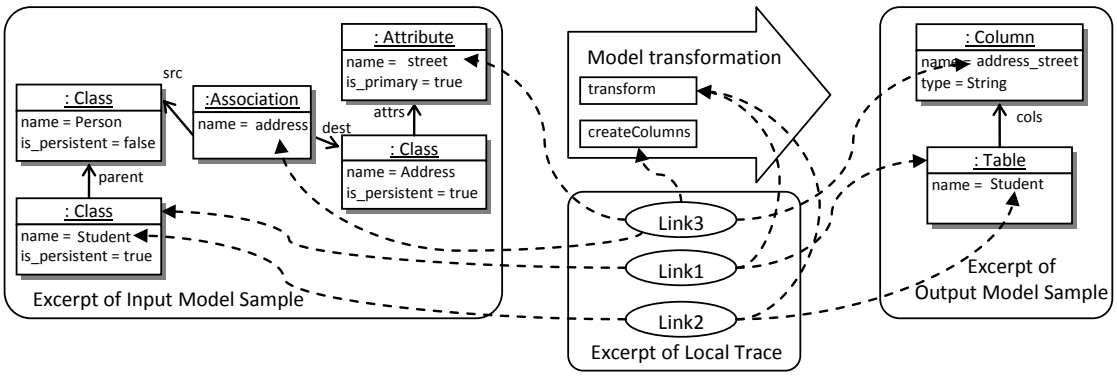


Fig. 8. Example of a trace between one input class model and its corresponding rdbms model

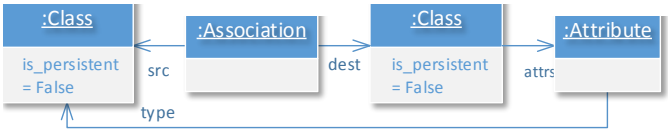


Fig. 9. Incorrect input pattern to prevent non-transformable models

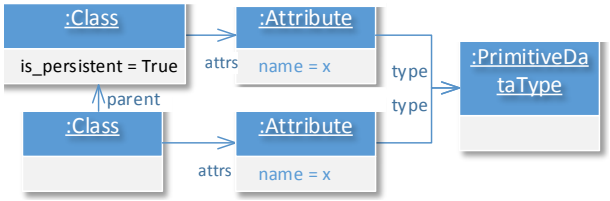


Fig. 10. Incorrect input pattern to prevent models to be transformed into incorrect output model

With this constraint, we prevent the creation of any input model containing such non transformable input pattern. This constraint identifies transitive closure through the type of an attribute *a1*, and an association *assoc1* in any input model. If this transitive closure exists, then the involved classes can't be both non persistent. This ALLOY predicate will then be used in PRAMANA generating test models for testing the model transformation. We can also add into the specification the pre-condition written in OCL:

```
context Association
dest.attrs->forAll(a | a.type == self.src
    implies (cl.is_persistent=True or a1.type.
    is_persistent=True) )
```

V. EVALUATION

We applied the proposed methodology on class2rdbms to discover new pre-conditions. The experiments are performed as an iterative process. During each iteration, we apply our methodology and discover *one new pre-condition*. We could detect several pre-conditions per iteration, however during the experiments we wanted to measure the improvement between two successive iterations. An improvement occurs when the number of incorrect input models decreases thanks to a new pre-condition. The first iteration has an **initial set of three pre-conditions**.

We applied the methodology twelve times, obtaining twelve new pre-conditions. Synthetically, as illustrated Figure 11 we rejected 580 models at the first iteration and only 22 after the last one.

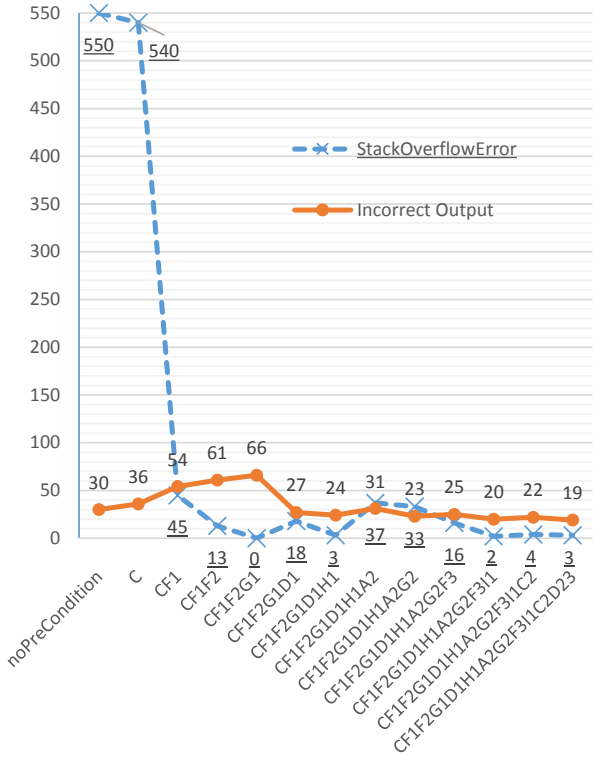


Fig. 11. Number of input models non-transformable or transformed into incorrect output models depending on the pre-conditions

A. Experiments set-up

We generate test models for class2rdbms using PRAMANA. We generate 10 non-isomorphic CD models (using symmetry breaking [33]) for each one of the 27 predicates. This give us a set of 270 test models.

We replicate the generation 7 times obtaining 7 sets of test models. Each set is generated using different factorial design parameters, as summarized in Table III. For each set we ask ALLOY to find models made of a different number of Class-Model, Classes, Associations, Attributes, PrimitiveDataTypes,

TABLE III. GENERATION DESIGN PARAMETERS FOR TEST MODEL GENERATION

Factors: Sets:	1	2	3	4	5	6	7
#ClassModel	1	1	1	1	1	1	1
#Class	5	5	10	10	5	10	5
#Association	5	10	5	10	5	5	10
#Attribute	25	25	25	25	30	30	30
#PrimitiveDataType	4	4	4	4	4	4	4
Bit-width Integer	5	5	5	5	5	5	5
#predicates	23	23	23	23	23	23	23
#models/predicates	10	10	10	10	10	10	10

and we give a specific range for the integer values of the properties. We obtained 1890 input models per iteration with diversity governed by the design parameters. We generate a total of 24,570 model, applying our methodology for pre-condition discovery 12 times and generating a final set to measure the final number of incorrect models (1890 * (12+1)).

B. Discovered Pre-conditions

We discovered 12 input patterns that lead to failure of execution of a transformation and we created 12 new pre-conditions, one in each iteration. We already explained in the *Approach* Section IV how we create the pre-condition *G1* (Section IV-C2) which was the fourth one we made.

The appendix A of the paper synthesizes the creation of two pre-conditions among the twelve ones we created. All the experimental material is available online [24].

C. Improvements

We drove the experiments trying to reduce the number of non-transformable models first, then the number of input models transformed into incorrect output models.

Thanks to the constraint *G1*, we had no more non-transformable models and the dashed curve is at 0. However, we observe that new non-executable models appear then. This is due to side effect of the new pre-condition created. It is dedicated to consider one incorrect input pattern, and once the new pre-condition is applied, this pattern does not get generated anymore. But preventing such pattern, it allows PRAMANA to explore into generating other "structures" in the models using the symmetry breaking scheme in ALLOY [33]. This leads to discovery of new incorrect patterns that we have to consider for newer pre-conditions.

While the 4 first pre-conditions (C, F1, F2, G1) (Figure 11 and Appendix A) reduces the number of non-executable models, they increase the number of incorrect output models. And when the 5th pre-condition (D1) reduces the number of incorrect output models, then the number of non-transformable models increases a bit.

If we consider globally the number of incorrect models, 10 among 12 pre-conditions improve the set of test models. The pre-condition A2 and C2 are the only ones which increase the number of non transformable models. However, they fully achieve their own objectives, because each one removes the incorrect pattern they were considering.

We observe that the addition of new pre-conditions continually leads to discovery of newer modeling patterns that are not handled by the initial specification [7] of class2rdbms produced by a group of human experts. However, we are unsure if this process of creating pre-conditions will terminate. We stop at 12 pre-conditions at the current state we still have 22 models that are not transformable correctly. Will extracting more pre-conditions from these models lead to pre-conditions that in turn lead to other types of strange and unforeseen patterns? Will this process every terminate? This is our challenge for the future.

VI. THREATS TO VALIDITY

Our framework for creating pre-conditions has 8 steps (Figure 5). While the first step is completely automatic, during our experiments the other steps still need human interaction. As explained before, generating a trace is complicated when the transformation crashed. The automation of the Step 5 and 6, identifying non transformable input excerpts, is in progress, without major difficulties. The refinement of the Step 7, producing incorrect input model patterns, can be done using EMF Compare, specializing the comparison. This step can be considered as automatic, even if we don't yet have a framework that would launch all those comparisons. Finally the Step 8 is manual now since if require domain and modeling experts contribution. However, since those pre-conditions will be part of the specification, it is better to ensure that they are validated by a tester.

Today there is no guarantee for the minimality and generalizability of the set of all pre-conditions. However, we empirically control pattern accuracy in the second pass of Step 7. Moreover, we can test that known correct models from previous iterations of pre-condition improvement to ensure that all models that were transformed into correct output models earlier always are in the new formulation of a pre-condition.

At the end of the 13 iterations, 22 input models are still incorrect. We could have continued to iterate. However, the transformation is complex (this is the reason why we use it as a running example), and could potentially involve chains of navigation through several different classes creating complex recursive closures. In our case we approximate this recursive closure constraint by stating a maximum depth considered in the pre-conditions. For instance, $F1$ and $F2$ constraints have depth 1 and 2 (Appendix A): when $F1$ prevents the creation of a cycle with one association, $F2$ prevents the creation of a cycle with two associations. It is not possible to use the Alloy transitive closure, such as the one of the Listing 1, because an *Association* is a class linked to the class *Class* through its two properties *src* and *dest* (Figure 2(a)). Another example is the 3 input models still non transformable at the end of the experiments. They require two new pre-conditions which are evolutions of the constraint C2.

Pre-conditions discovery is done in our experiments assuming a *correct model transformation*. In that case, every new pre-condition rejects failed inputs as being out of the input domain. When a new pre-condition completes the specification by more precisely specifying the transformation behavior is not yet considered. First, it implies considering the testing and debugging of the transformation. Second, it is more complicated to create a pre-condition in Alloy or OCL when it should describe a transformation behavior depending on input model properties. Both these cases will be considered in future work.

This difficulty to create some pre-conditions can be circumvented by implementing the pre-condition into the transformation as a rule/operation. Using an imperative language such as Kermeta allows it. Moreover, this limitation will complicate the testing, preventing it's automation, but the tester can consider those constraints as soon as they are described textually even if they are not implemented. For this reason, we believe that our contribution is achieved as soon as the pre-condition is described, and even further when we can implement it.

We considered a model transformation implemented in Kermeta, nevertheless the approach is not language dependent.

VII. Related Work

We have considered the testing of model transformations previously. After identifying its challenges [6], we have listed model transformations' main operations that should be tested [25]. We successfully considered the partitioning of its input domain when testing [15]. In [9] we present an automated generation technique for models that conform only to the Ecore diagram of a meta-model specification. Finally, the tool PRAMANA consider the meta-model specification with all its constraints [30]. We have already used PRAMANA to test model transformation based on static analysis [26].

Accurately specifying the input domain of a model transformation is the problem addressed in this paper. The object constraint language (OCL) [12] is the standard to specify model transformation pre-conditions to prevent it from processing input models outside its true input domain. Model transformation developers find it intuitive to transform specific patterns in an input model to elements of the output model such as by example [35] or demonstration [34]. In [36], the authors go a step further and automate model transformation by example using inductive logic programming. In [21] the authors use a search-based technique to find transformations (from a repository) for example input models. Model transformation executability has also been explored in [11], where the authors verify transformation executability by solving pre and post conditions as a constraint program. Deciding on correct transformation rules for an input model such that its transformation is efficient and does not appear to loop is a challenge addressed in [19]. The authors present an approach based on dynamic scope discovery using a naive bayes classifier trained with rules mapped to input modeling patterns. To the best of our knowledge, the literature on specifying model transformations discusses little about identifying and handling modeling patterns that *may not have* been foreseen by a transformation developer. This set of input models is infinite and its impact on the transformation is unknown.

In this article, we systematically explore the potentially infinite input domain to specify a model transformation's contracts (pre-condition in particular). We generate models that can break a model transformation's execution by running into an infinite loop or violating a post-condition. Model generation is more general and complex than generating integers, floats, strings, lists, or other standard data structures such as dealt with in the Korat tool of Chandra et al. [8]. Korat is faster than ALLOY in generating data structures such as binary trees, lists, and heap arrays from the Java Collections Framework but it does not consider the general case of models which are arbitrarily constrained graphs of objects. The constraints on models makes model generation a different problem than generating test suites for context-free grammar-based software [22] which do not contain domain-specific constraints. Test models are complex graphs that must conform to an input meta-model specification, a transformation pre-condition and additional knowledge such as model fragments to possibly reveal violation of a post-condition or run into an infinite loop. Such a methodology using graph transformation rules is presented in [13]. Generated models in both these approaches do not satisfy the constraints on the meta-model. In [31] we presented a method to generate models given partial models by transforming the meta-model and partial model to a Constraint Logic Programming (CLP). We solve the resulting CLP to give model(s) that conform to the input domain. However, the approach does not add new objects to the model. We assume that the number and types of models in the partial model is sufficient for obtaining complete models. The constraints in this system are limited to first-order horn clause logic.

VIII. Conclusion

In this paper we presented an approach to discover new pre-conditions completing a model transformation's specification. We use traceability from automatically generated test models to identify such pre-conditions. The eight-step approach helps identify incorrect input patterns in numerous and large input models. Those patterns are transformed into pre-conditions preventing input models to loop infinitely or to be transformed into incorrect output models. The experimental results show that this semi-automated approach produces pre-conditions drastically reducing the number of incorrect input models. The paper can be also interpreted as evidence of the fact that the specification of a model transformation is quite complicated to write completely, and that such a dynamic approach helps improve its completeness.

The new pre-conditions are added to the specification of a model transformation. This has helped improve specification to help a tester find bugs in the implementation and improve it. He/she can choose how the pre-conditions are used:

- He/she can choose to use the pre-condition to reduce the input domain, rejecting input models not supposed to be transformed.
- He/she can embed the pre-condition in the implementation with a behavior considering the input models' incorrectness. For instance, it can raise an exception which will be caught properly without crashing the transformation (e.g. division by zero should raise an *ArithmeticException*), or implement transformation rules to process this incorrect part of an input model (e.g. ignoring this part in the output model for instance).

In future, we plan to automate generation of concise and effective ALLOY predicates directly from traces to feedback the model generation process itself. For instance, in [32], the authors have automated the completion of partial models specified in a model editor by generating ALLOY predicates from a partial model. Finally, we explore a general completion criteria to discover all possible pre-conditions in a transformation given a set of trace links as input (Figure 8) in a feedback loop of executing a transformation with every refinement of its input domain.

REFERENCES

[1] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. On challenges of model transformation from uml to alloy. In *Software and Systems Modeling*, volume 9, pages 69–86, December 2009.

[2] Vincent Aranega, Anne Etien, and Jean-Luc Dekeyser. Using an alternative trace for qvt. In *Workshop on Multi-Paradigm Modeling*, 2010.

[3] Vincent Aranega, Jean-Marie Mottu, Anne Etien, Thomas Degueule, Benoit Baudry, and Jean-Luc Dekeyser. Towards an automation of the mutation analysis dedicated to model transformation. *Software Testing, Verification and Reliability*, 25(5-7):653–683, 2015.

[4] Thomas Baar. The definition of transitive closure with ocl-limitations and applications. In *Proceedings of Fifth Andrei Ershov International Conference, Perspectives, of System Informatics*, pages 358–365. Springer, 2003.

[5] R. Bardohl, G. Taentzer, and A. Schurr M. Minas. *Handbook of Graph Grammars and Computing by Graph transformation, vII: Applications, Languages and Tools*. World Scientific, 1999.

[6] Benoit Baudry, Sudipto Ghosh, Franck Fleurey, Robert France, Yves Le Traon, and Jean-Marie Mottu. Barriers to systematic model transformation testing. *Communications of the ACM*, 53(6), 2010.

[7] Jean Bezivin, Bernhard Rumpe, Andy Schurr, and Laurence Tratt. Model transformations in practice workshop, october 3rd 2005, part of models 2005. In *Proceedings of MoDELS*, 2005.

[8] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. In *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, 2002.

[9] E. Brottier, F. Fleurey, J. Steel, B. Baudry, and Y. Le Traon. Metamodel-based test generation for model transformations: an algorithm and a tool. In *Proceedings of ISSRE'06*, Raleigh, NC, USA, 2006.

[10] Frank Budinsky. *Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley, 2004.

[11] Jordi Cabot, Robert Clarisó, and Daniel Riera. Verifying UML/OCL operation contracts. In *Integrated Formal Methods*, pages 40–55. Springer, 2009.

[12] Eric Cariou, Raphaël Marvie, Lionel Seinturier, and Laurence Duchien. OCL for the specification of model transformation contracts. In *OCL and Model Driven Engineering, UML 2004 Conference Workshop*, volume 12, pages 69–83, 2004.

[13] K. Ehrig, J.M. Küster, G. Taentzer, and J. Winkelmann. Generating instance models from meta models. In *FMOODS'06 (Formal Methods for Open Object-Based Distributed Systems)*, pages 156 – 170., Bologna, Italy, June 2006.

[14] Jean-Rémy Falleri, Marianne Huchard, and Clémentine Nebut. Towards a traceability framework for model transformations in kermeta. In *ECMDA-TW Workshop*, 2006.

[15] Franck Fleurey, Benoit Baudry, Pierre-Alain Muller, and Yves Le Traon. Qualifying input test data for model transformations. *Software & Systems Modeling*, 8(2):185–203, 2009.

[16] Jean-Marc Jézéquel and Bertrand Meyer. Design by contract: The lessons of ariane. *Computer*, 30(1):129–130, 1997.

[17] Frédéric Jouault. Loosely coupled traceability for ATL. In *ECMDA Workshop on Traceability*, Germany, 2005.

[18] Frédéric Jouault and Ivan Kurtev. On the Architectural Alignment of ATL and QVT. In *Proceedings of ACM Symposium on Applied Computing (SAC 06)*, Dijon, FRA, April 2006.

[19] Māris Jukšs, Clark Verbrugge, Dániel Varró, and Hans Vangheluwe. Dynamic scope discovery for model transformations. In *Software Language Engineering*, pages 302–321. Springer, 2014.

[20] Kermeta. http://www.kermeta.org/.

[21] Marouane Kessentini, Houari Sahraoui, Mounir Boukadoum, and Omar Ben Omar. Search-based model transformation by example. *Software & Systems Modeling*, 11(2):209–226, 2012.

[22] Hennessy M and J.F. Power. An analysis of rule coverage as a criterion in generating minimal test suites for grammar-based software. In *Proc. of the 20th IEEE/ACM ASE*, NY, USA, 2005.

[23] Bertrand Meyer. Applying'design by contract'. *Computer*, 25(10):40–51, 1992.

[24] Jean-Marie Mottu. ISSRE 2015 experiments material. http://pagesperso.lina.univ-nantes.fr/~mottu-jm/development-en.html, 2015.

[25] Jean-Marie Mottu, Benoit Baudry, and Yves Le Traon. Mutation analysis testing for model transformations. In *Proceedings of ECMDA'06*, Bilbao, Spain, July 2006.

[26] Jean-Marie Mottu, Sagar Sen, Massimo Tisi, and Jordi Cabot. Static analysis of model transformations for effective test generation. In *IEEE International Symposium on Software Reliability Engineering, ISSRE 2012*, Dallas, USA, November 2012. IEEE.

[27] Pierre-Alain Muller, Franck Fleurey, and Jean-Marc Jézéquel. Weaving executability into object-oriented meta-languages. In *MODELS/UML*, volume 3713, pages 264–278, Montego Bay, Jamaica, October 2005. Springer.

[28] OMG. Mof 2.0 core specification. Technical Report formal/06-01-01, OMG, April 2006. OMG Available Specification.

[29] OMG. The Object Constraint Language Specification 2.0, OMG Document: ad/03-01-07, 2007.

[30] Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. On combining multi-formalism knowledge to select test models for model transformation testing. In *IEEE International Conference on Software Testing*, Lillehammer, Norway, April 2008.

[31] Sagar Sen, Benoit Baudry, and Doina Precup. Partial model completion in model driven engineering using constraint logic programming. In *International Conference on the Applications of Declarative Programming*, 2007.

[32] Sagar Sen, Benoit Baudry, and Hans Vangheluwe. Towards domain-specific model editors with automatic model completion. *Simulation*, 86(2):109–126, 2010.

[33] Ilya Shlyakhter. Generating effective symmetry-breaking predicates for search problems. *Discrete Applied Mathematics*, 155(12):1539–1548, 2007.

[34] Yu Sun, Jules White, and Jeff Gray. Model transformation by demonstration. In *Model Driven Engineering Languages and Systems*, pages 712–726. Springer, 2009.

[35] Dániel Varró. Model transformation by example. In *Model Driven Engineering Languages and Systems*, pages 410–424. Springer, 2006.

[36] Dániel Varró and Zoltán Balogh. Automating model transformation by example using inductive logic programming. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 978–984. ACM, 2007.

APPENDIX

All of the experiments material is available online [24]. Here we list 2 new Alloy Preconditions $F1$ and $F2$, used to illustrate the Section VI. We also illustrate 2 iterations among the 12 ones that we processed producing 12 new pre-conditions.

A. Pre-conditions F1 and F2 illustrating transitive closure of different depths

```
// F1. No association loop between one non-
    persistent class

fact no1CycleNonPersistent{
all a: Association | (a.dest == a.src) => a.
    dest.is_persistent= True
}

// F2. No association loop between 2 non-
    persistent classes

fact no2CycleNonPersistent{
all a1: Association, a2:Association | (a1.dest
     == a2.src and a2.dest==a1.src) => a1.src.
    is_persistent= True or a2.src.is_persistent
    =True
}
```

B. Fourth Iteration: creating pre-condition G1

1) Identified non-transformable input models: Here is part of the list of the non-transformable models which have been identified during the fourth iteration.

i. model 7 created with the 4th predicate, using AllRanges strategy (mfAllRanges4), in the set 1
ii. model 4 created with the 4th predicate, using AllRanges strategy (mfAllRanges4), in the set 4
iii. model 9 created with the 3rd predicate, using AllRanges strategy (mfAllRanges3), in the set 6
iv. etc.

2) Model of the trace: We used the trace illustrated Figure 12 to identify the non-transformable excerpt from the model iii. The bottom part of the model is the LocalTrace [2]. It lists *traceabilityLinks* referring the *sourceElements* which are transformed. We do not illustrate the transformation rules (not used in this work) and the output model element since the transformation doesn't return any. The transformation loops on a series of two links (two on the left, then two on the right, etc.). We used them identifying the non-transformable excerpt of the Figure 4, p.5.

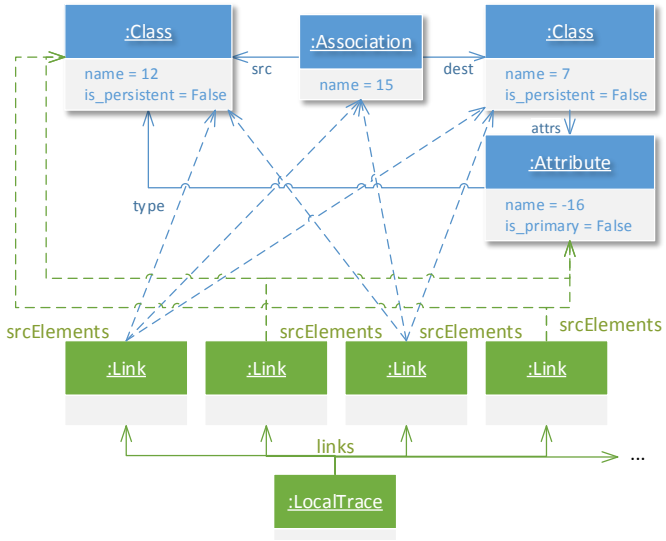


Fig. 12. Part of the model of the trace used to identify non-transformable input excerpt from a non-transformable input model

3) Identified non-transformable input excerpt: Illustrated Figure 4, p.5.

4) Refined incorrect input pattern: Illustrated Figure 9, p.8.

5) New pre-condition G1 preventing non-transformable input models: The pre-condition *G1* specifies that it should be no loop type - association between 2 non-persistent classes. It is listed in Section IV-C2

C. Fifth Iteration: creating pre-condition D1

1) Identified incorrect output models: During the fifth iteration, we identified a set of incorrect output models which have been generated from incorrect input models:

I. model number 4 created with the Predicate 3, using AllPartitions strategy (mfAllPartitions3), in the set 1
II. model number 2 created with the Predicate 3, using AllPartitions strategy (mfAllPartitions3), in the set 1
III. model number 4 created with the Predicate 6, using AllRanges strategy (mfAllRanges6), in the set 6
IV. etc.

2) Model of the trace: We used the trace illustrated Figure 13 to identify the non-transformable input excerpt from the incorrect output model I.

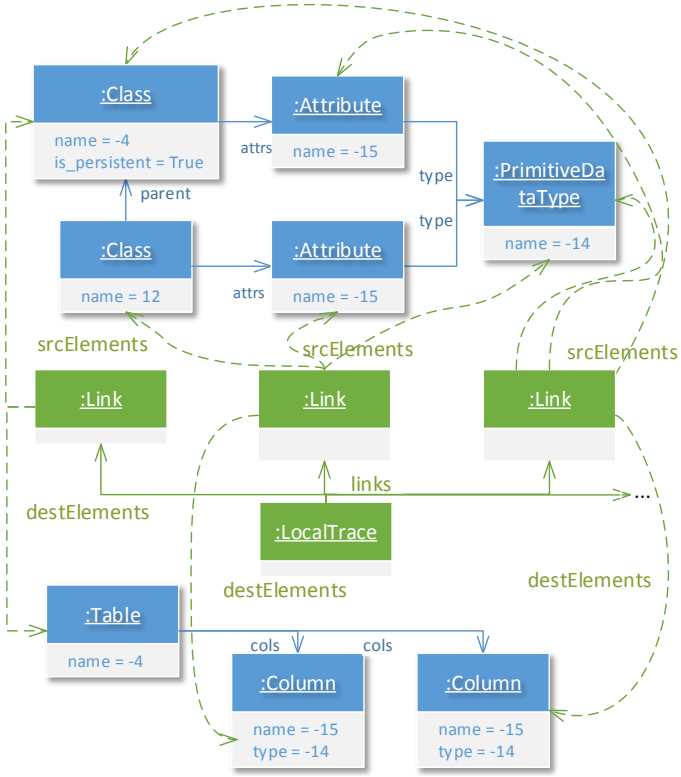


Fig. 13. Part of the model of the trace used to identify the incorrect input excerpt from an incorrect output model

3) Identified incorrect input excerpt producing incorrect output model: Illustrated Figure 6, p.6.

4) Refined incorrect input pattern: Illustrated Figure 10, p.8.

5) New pre-condition D1 preventing generating incorrect output models: Listed in in Alloy and in OCL:

```
// D1. A class A which inherits from a
    persistent class B can't have an attribute
    with the same name and type that one
    attribute of that persistent class B
fact classInheritsOutgoingNotSameNameAttrib
{
  all A:Class | all B:A.^parent | B.
      is_persistent == True implies (no a1: A.
      attrs, a2:B.attrs | (a1.name=a2.name and
      a1.type=a2.type))
}

context Class
self.allParents() -> includes(c1 | c1.
    is_persistent) implies self.attrs -> forAll
    (a1 | c1.attrs -> select(a2 |a1.name = a2.
    name) -> isEmpty())
```